Static Analysis Alert Audits
Lexicon And Rules

William Snavely

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM-0004189

Audit Lexicon And Rules

Background

Lexicon

Rules

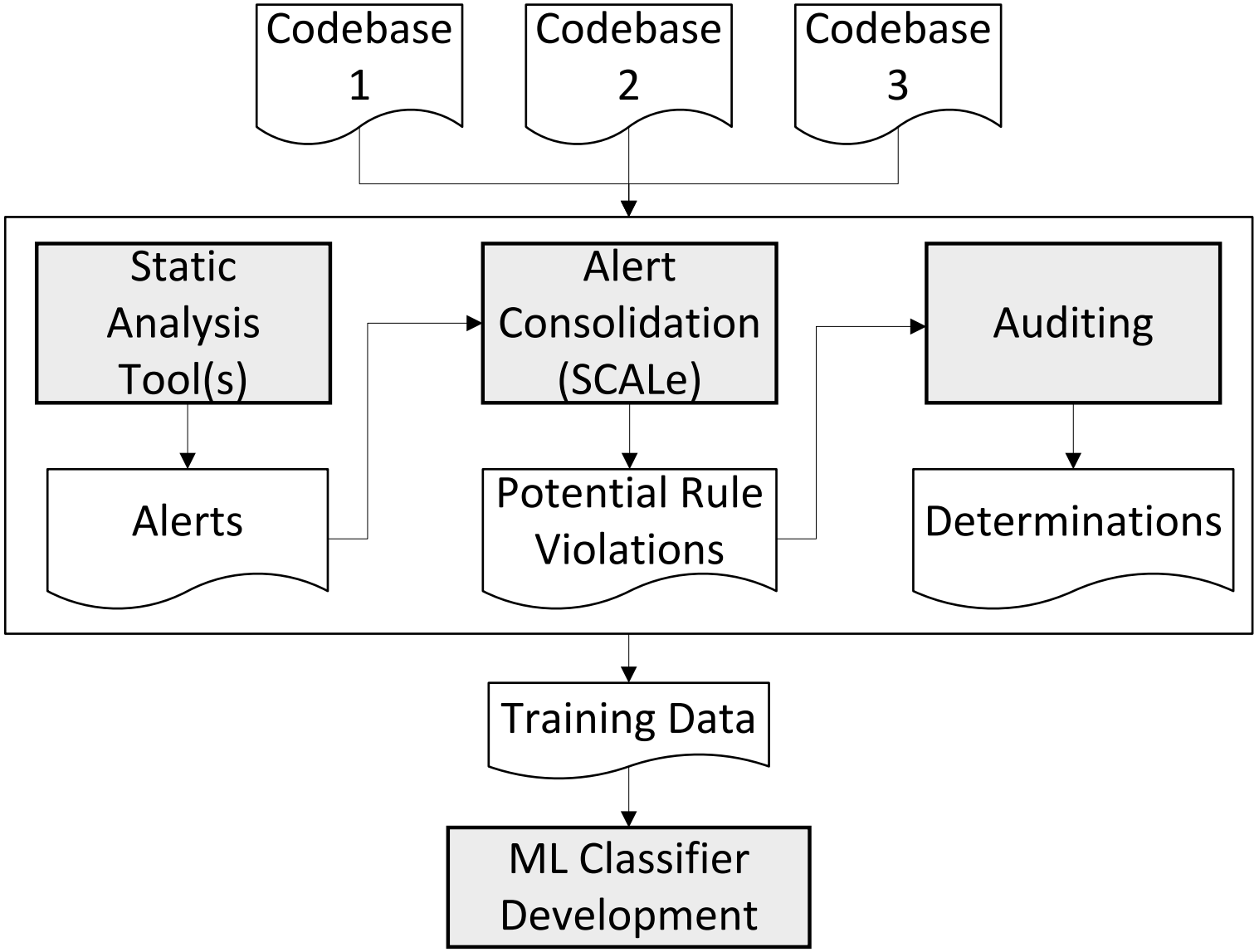
Future Work

Questions?

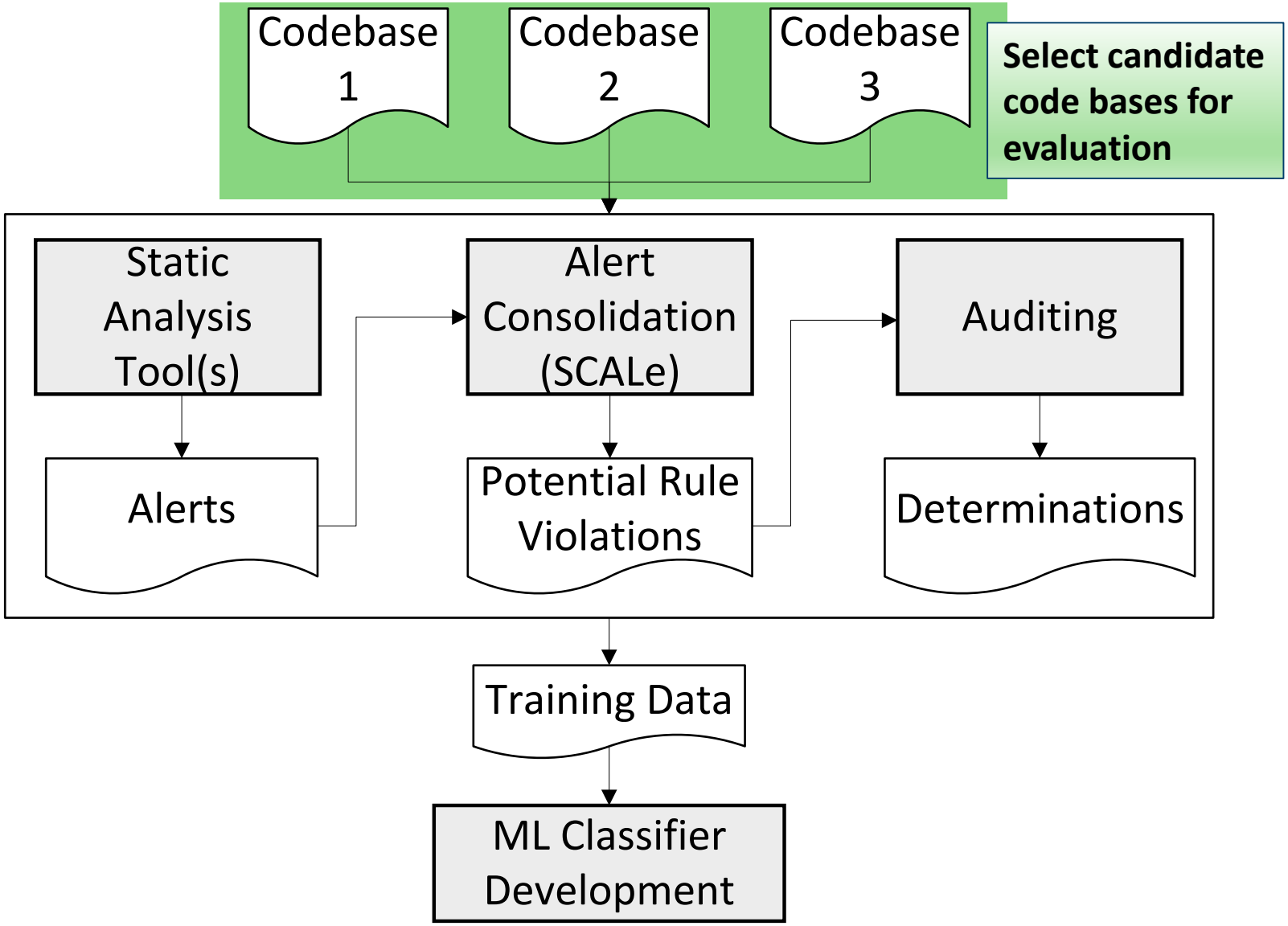
Audit Lexicon And Rules

Background

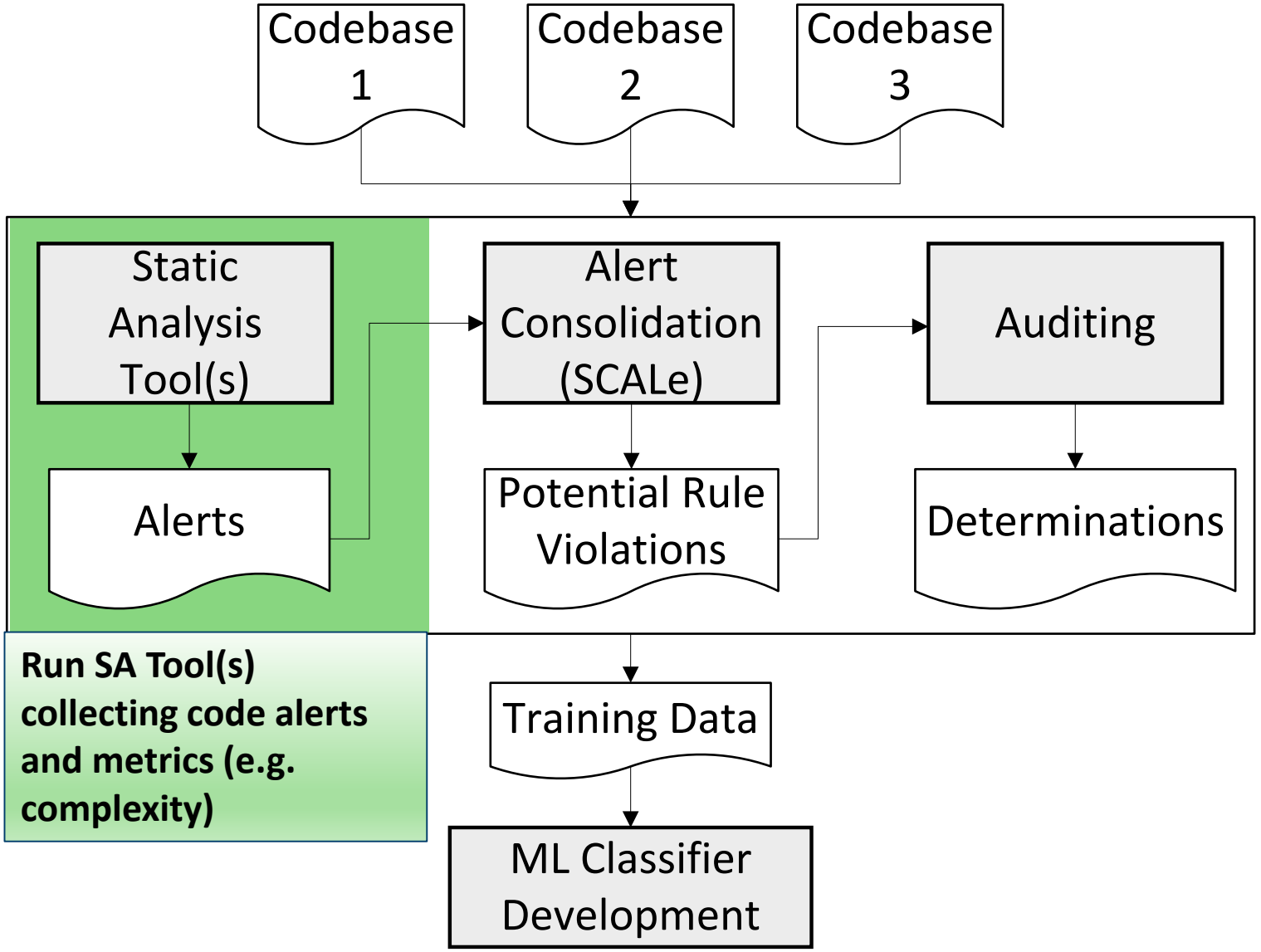
Background: Automatic Alert Classification



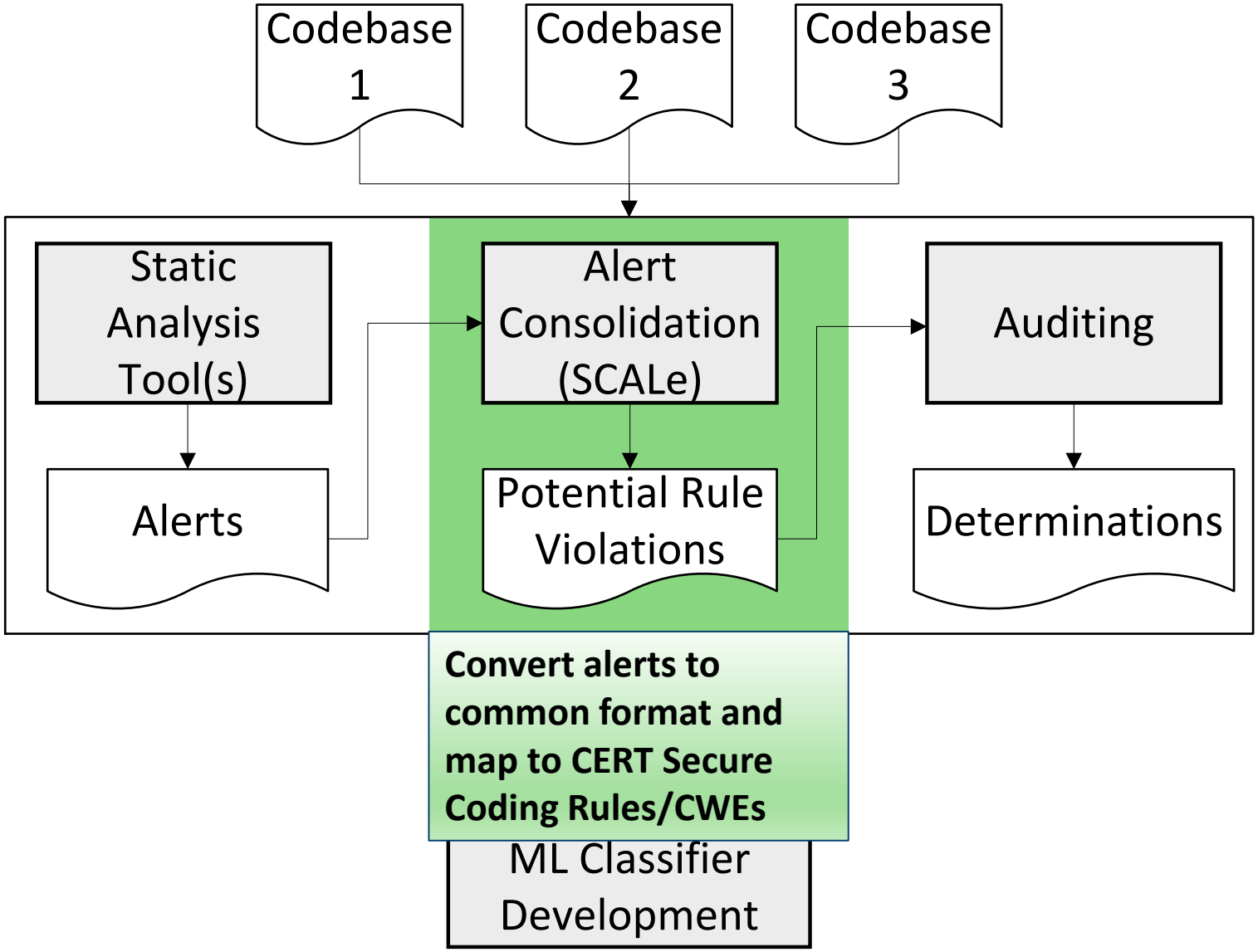
Background: Automatic Alert Classification



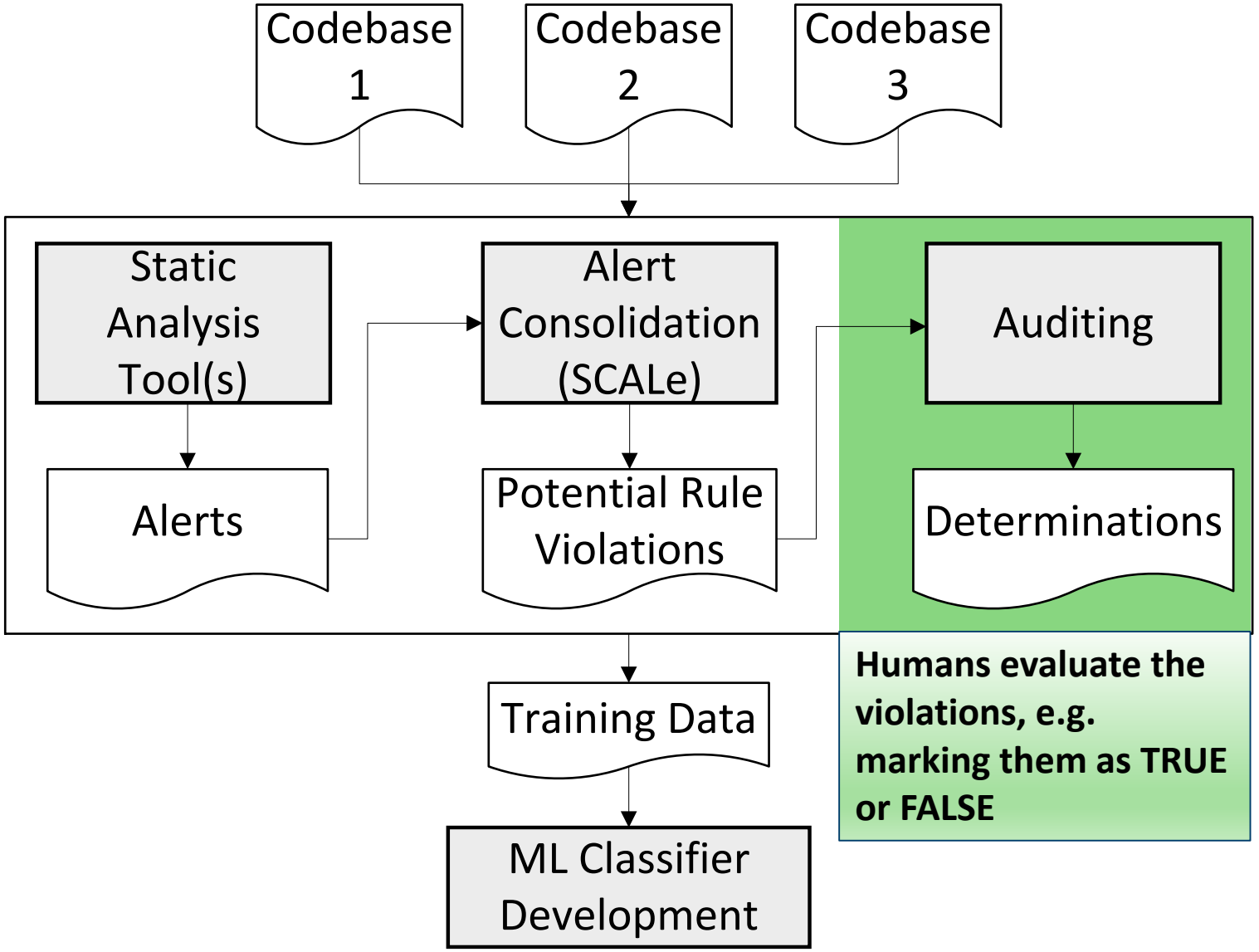
Background: Automatic Alert Classification



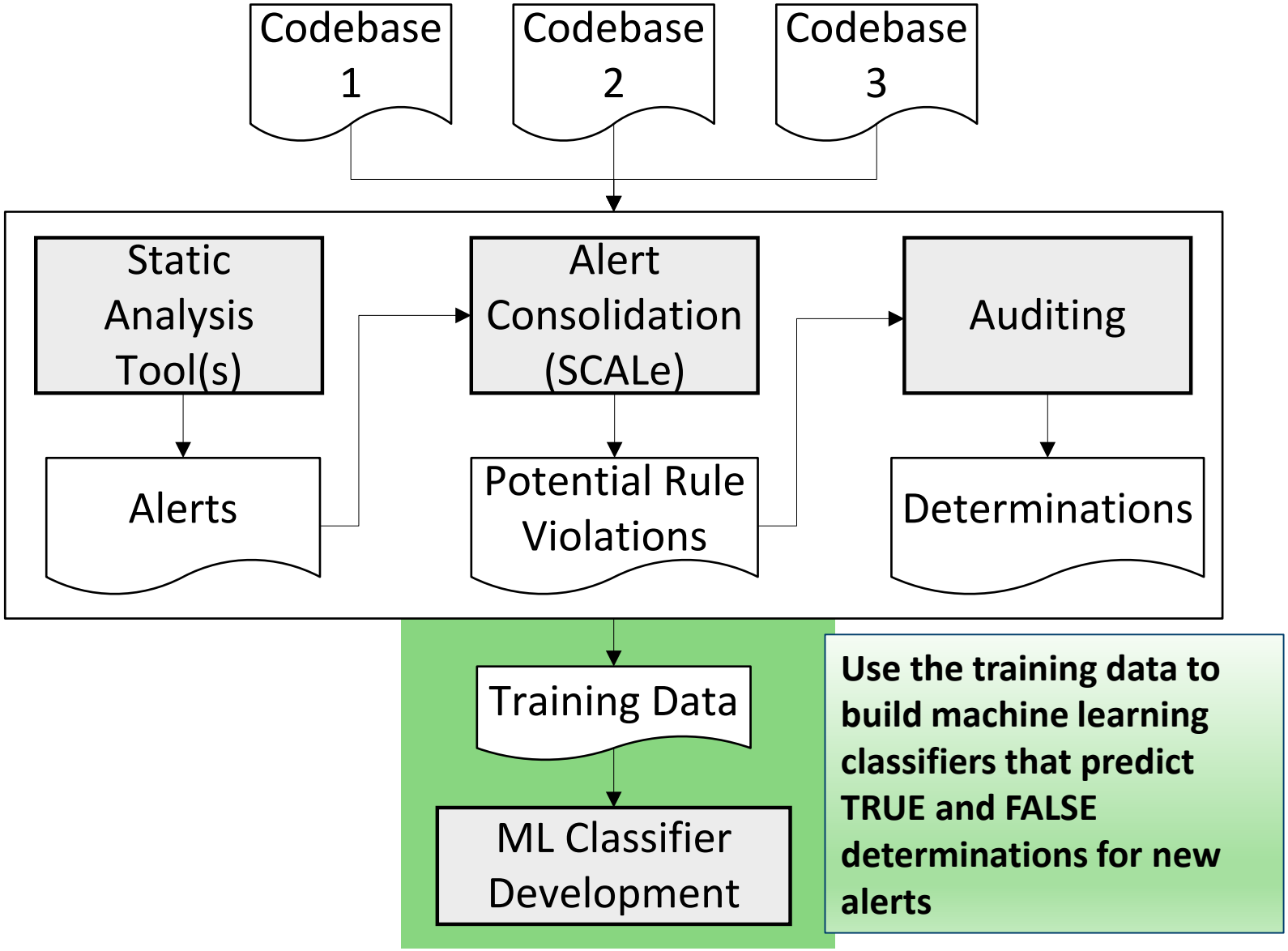
Background: Automatic Alert Classification



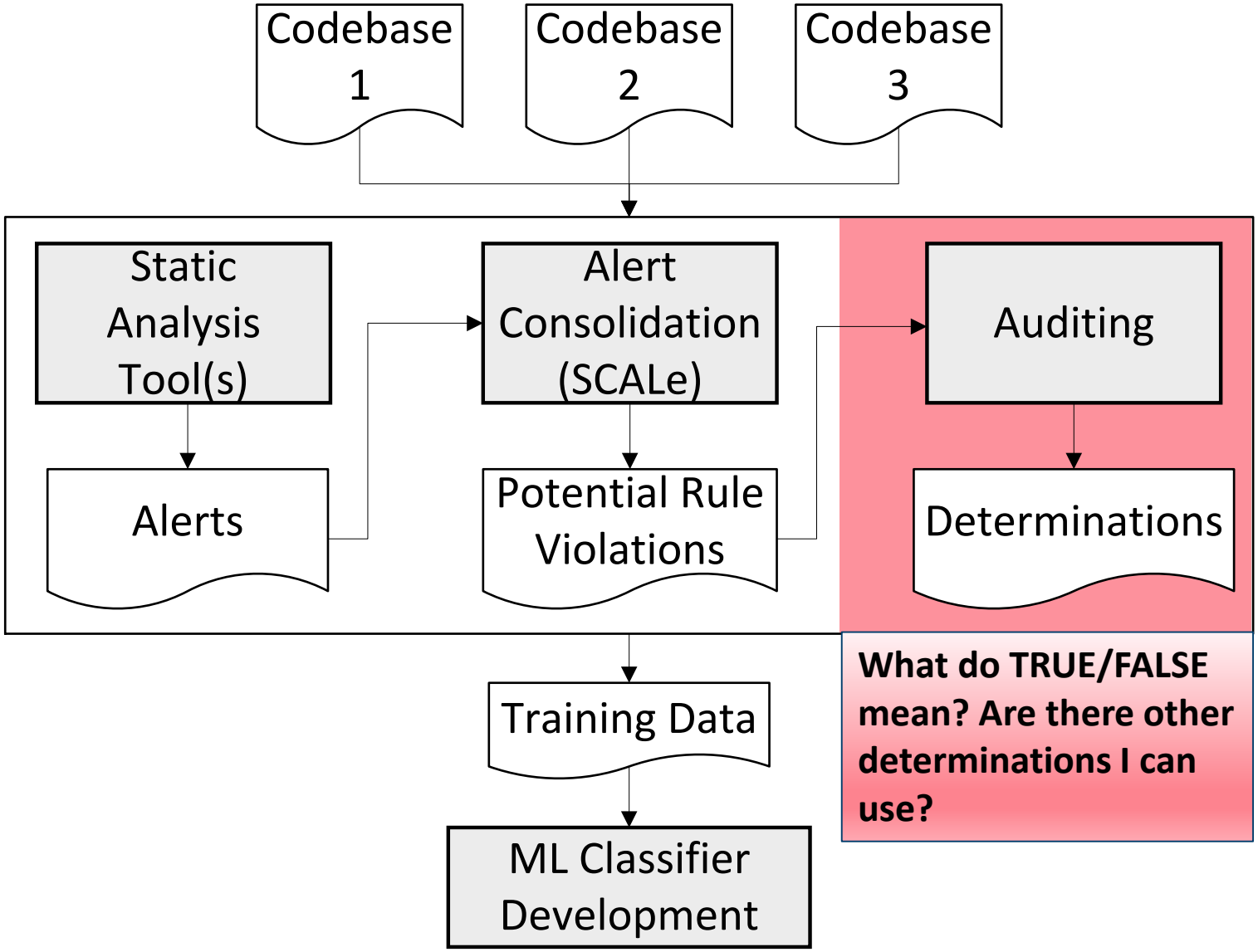
Background: Automatic Alert Classification



Background: Automatic Alert Classification



Background: Automatic Alert Classification

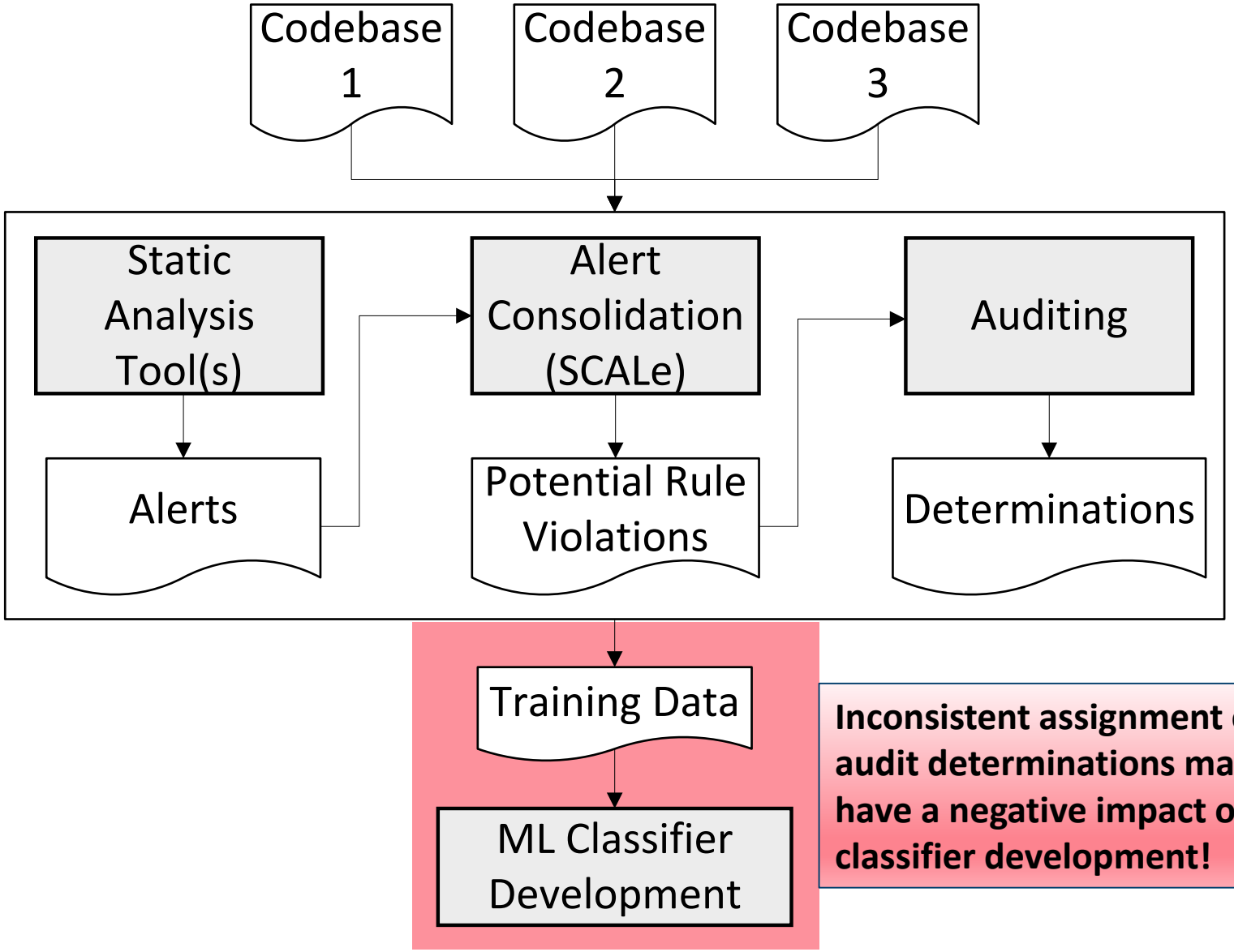


What is truth?

One collaborator reported using the determination **True** to indicate that the issue reported by the alert was a real problem in the code.

Another collaborator used **True** to indicate that *something* was wrong with the diagnosed code, even if the specific issue reported by the alert was a **false positive**!

Background: Automatic Alert Classification



Inconsistent assignment of audit determinations may have a negative impact on classifier development!

Solution: Lexicon And Rules

- We developed a **lexicon** and auditing **rule set** for our collaborators
- Includes a standard set of well-defined **determinations** for static analysis alerts
- Includes a set of **auditing rules** to help auditors make consistent decisions in commonly-encountered situations

Different auditors should make the **same determination** for a given alert!

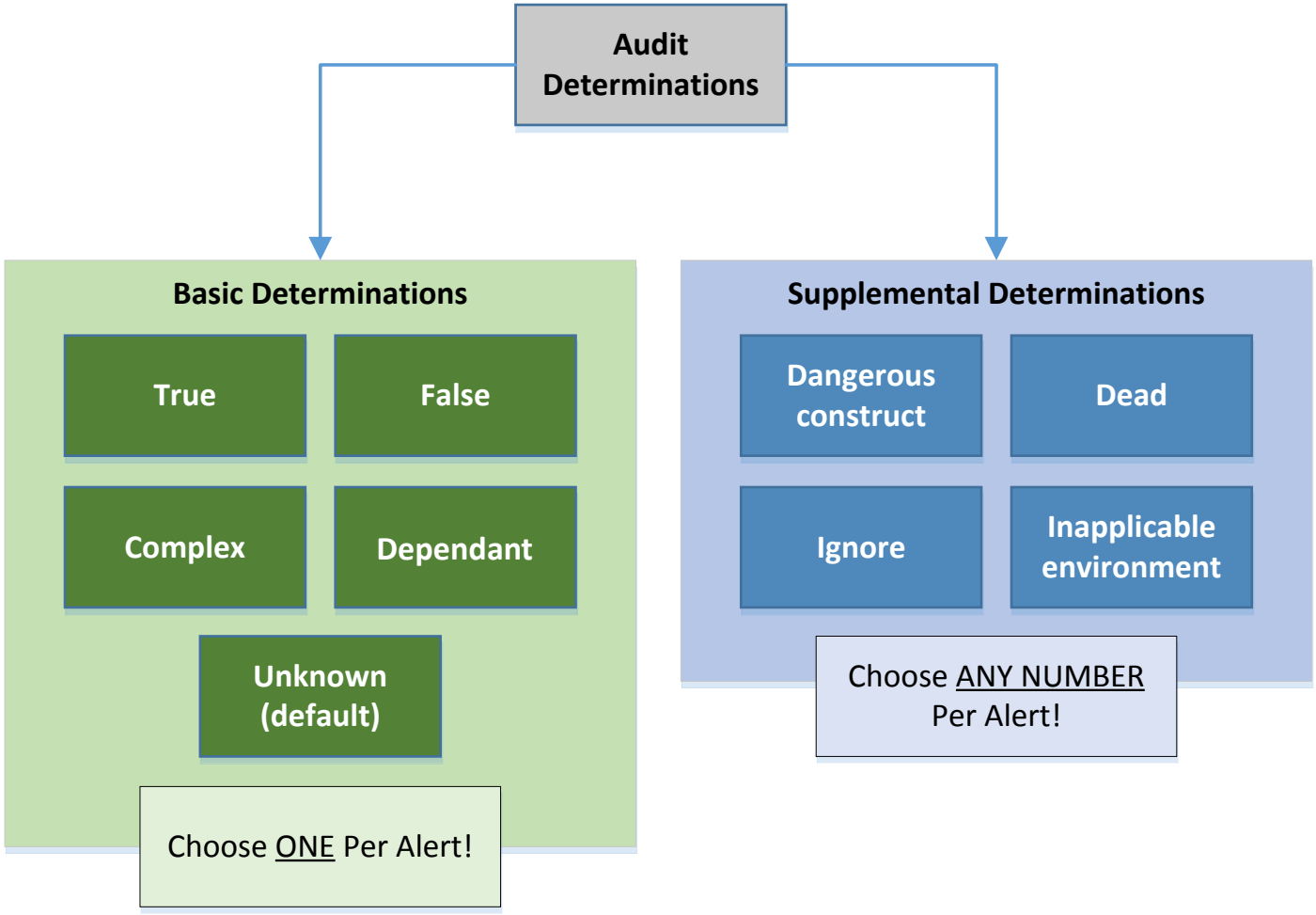
Improve the **quality and consistency** of audit data for the purpose of building **machine learning classifiers**

Help organizations make **better-informed** decisions about **bug-fixes**, **development**, and **future audits**.

Audit Lexicon And Rules

Lexicon

Lexicon: Audit Determinations



Lexicon: Basic Determinations

True

- The code in question violates the **condition** indicated by the alert.
- A **condition** is a constraint or property of validity.
  - E.g. A valid program should not deference NULL pointers.
- The condition can be determined from the definition of the alert itself, or from the **coding taxonomy** the alert corresponds to.
  - CERT Secure Coding Rules
  - CWEs

Lexicon: Basic Determinations
True Example

```
char *build_array(size_t size, char first) {
        if(size == 0) {
                return NULL;
        }

        char *array = malloc(size * sizeof(char));
        array[0] = first;
        return array;
}
```

ALERT: Do not dereference NULL pointers!

Determination: **TRUE**

Lexicon: Basic Determinations

False

- The code in question does **not** violate the **condition** indicated by the alert.

```
char *build_array(int size, char first) {
        if(size == 0) {
                return NULL;
        }

        char *array = malloc(size * sizeof(char));
        if(array == NULL) {
                abort();
        }
        array[0] = first;
        return array;
}
```

Determination: **FALSE**

ALERT: Do not dereference NULL pointers!

Lexicon: Basic Determinations

Complex

- The alert is **too difficult** to judge in a **reasonable amount of time and effort**
- "Reasonable" is defined by the individual organization.

Dependent

- The alert is related to a **True** alert that occurs earlier in the code.
- Intuition: fixing the first alert would implicitly fix the second one.

Unknown

- None of the above. This is the default determination.

Lexicon: Basic Determinations Dependent Example

```
char *build_array(size_t size, char first, char last) {
        if(size == 0) {
                return NULL;
        }

        char *array = malloc(size * sizeof(char));
        array[0] = first;
        array[size - 1] = last;
        return array;
}
```

ALERT: Do not dereference NULL pointers!

Determination: **TRUE**

ALERT: Do not dereference NULL pointers!

Determination: **DEPENDENT**

Lexicon: Supplemental Determinations

Dangerous Construct

- The alert refers to a piece of code that poses **risk** if it is not modified.
- Risk level is specified as **High**, **Medium**, or **Low**
- Independent of whether the alert is true or false!

Dead

- The code in question **not reachable at runtime.**

Inapplicable Environment

- The alert does not apply to the current environments where the software runs (OS, CPU, etc.)
- If a new environment were added in the future, the alert may apply.

Ignore

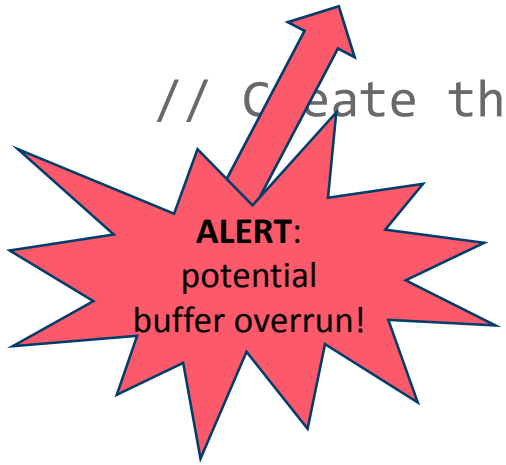
- The code in question does not require mitigation.

Lexicon: Supplemental Determinations Dangerous Construct Example

```
#define BUF_MAX 128

void create_file(const char *base_name) {
        // Add the .txt extension!
        char filename[BUF_MAX];
        snprintf(filename, 128, "%s.txt", base_name);

        // Create the file, etc...
}
```



Seems ok…but why not use **BUF_MAX** instead of 128?

Determination:
False
+
Dangerous Construct

Audit Lexicon And Rules

Rules

Audit Rules

Goals

- Clarify **ambiguous or complex** auditing scenarios
- Establish **assumptions** auditors can make
- Overall: help make audit determinations **more consistent**

We developed **12 rules**

- Drew on our own experiences auditing code bases at CERT
- Trained 3 groups of engineers on the rules, and incorporated their feedback
- In the following slides, we will inspect three of the rules in more detail.

Example Rule: Assume external inputs to the program are malicious

An auditor should assume that **inputs to a program module** (e.g. function parameters, command line arguments, etc.) may have arbitrary, **potentially malicious**, values.

- Unless they have a strong guarantee to the contrary

Example from recent history: **Java Deserialization**

- Suppose an alert is raised for a call to `readObject`, citing a violation of the CERT Secure Coding Rule **SER12-J, Prevent deserialization of untrusted data**
- An auditor can assume that external data passed to the `readObject` method may be malicious, and mark this alert as **True**
  - Assuming there are no other mitigations in place in the code

Audit Rules
External Inputs Example

```
import java.io.*;

class DeserializeExample {
    public static Object deserialize(byte[] buffer)
            throws Exception {
        ByteArrayInputStream bais;
        ObjectInputStream ois;
        bais = new ByteArrayInputStream(buffer);
        ois = new ObjectInputStream(bais);
        return ois.readObject();
    }
}
```

ALERT: Don't deserialize untrusted data!

Without strong evidence to the contrary, assume the buffer could be malicious!

Determination: **TRUE**

Example Rule: Unless instructed otherwise, assume code must be portable.

When auditing alerts for a code base where the target platform is **not specified**, the auditor should **err on the side of portability**.

If a diagnosed segment of code **malfunctions on certain platforms**, and in doing so violates a condition, this is suitable justification for marking the alert **True**.

Audit Rules
Portability Example

```
int strcmp(const char *str1, const char *str2) {
      while(*str1 == *str2) {
            if(*str1 == '\0') {
                  return 0;
            }
            str1++;
            str2++;
      }

      if(*str1 < *str2) {
            return -1;
      } else {
            return 1;
      }
}
```

ALERT: Cast to **unsigned char** before comparing!

This code would be safe on a platform where chars are unsigned, but that hasn't been guaranteed!

Determination: **TRUE**

Example Rule: Handle an alert in unreachable code depending on whether it is exportable.

Certain code segments may be **unreachable** at runtime. Also called **dead code.**

A static analysis tool might not be able to realize this, and **still mark alerts** in code that **cannot be executed**.

The **Dead** supplementary determination can be applied to these alerts.

However, an auditor should **take care** when deciding if a piece of code is truly dead.

In particular: just because a given program module (function, class) is not used does **not** mean it is dead. The module might be exported as a **public interface**, for use by another application.

This rule was developed as a result of a scenario encountered by one of our collaborators!

Future Work

- Gather feedback on our lexicon and rules from surveys, focus groups, experts, etc.
- Continue to refine the lexicon/rules.
- Further develop CERT's SCALe auditing framework to fully incorporate these concepts.
- Work with more collaborators to test the rules/lexicon in practice.
  - We have some initial feedback from two collaborators, who used our rules to audit several hundred alerts from C and Java codebases

Audit Lexicon And Rules

Questions?